FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Performance Analysis for SVM-Fortran with OPAL

M. Gerndt, A. Krumme, S. Özmen

KFA-ZAM-IB-9519

August 1995
(Stand 23.08.95)

Appears in:
Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), November 3-4, 1995, Athens, Georgia, USA

Performance Analysis for SVM-Fortran with OPAL

M. Gerndt, A. Krumme, S. Özmen
Research Centre Jülich (KFA)
Central Institute for Applied Mathematics
52425 Jülich, Germany
Phone: ++49 2461 616569
Fax: ++49 2461 616656
{m.gerndt, a.krumme, s.oezmen}@kfa-juelich.de

Abstract

Programming distributed memory parallel computers with message passing is often considered to be a difficult task. To overcome the drawbacks of this programming style, several efforts have been made in the field of new parallel programming languages. One example is the shared virtual memory programming model. SVM-Fortran is an extension of Fortran77 for programming shared virtual memory systems. It provides special notations for work distribution to optimize data locality and load balance. There exists a number of different tools for performance debugging in message passing systems, but none of these would fulfill the special requirements of the SVM programming model. Moreover, we observed that the user requirements for performance debugging strongly recommend tools which are adapted to the programmers view of the program. Therefore, special emphasis is set on source code related methods of OPAL and the underlying trace generation with the performance monitor SAM.

Keywords: distributed memory computers, shared virtual memory, parallel programming, data locality, trace generation, performance analysis

1 Introduction

Programming distributed memory parallel computers with message passing is often considered to be a difficult task. Several approaches to simplify this task are being pursued such as HPF, shared virtual memory (SVM), or Linda. Common to these approaches is the emulation of a global address space by software, either by the compiler, the operating system, or the language runtime system.

Shared virtual memory systems provide a global address space on the basis of the virtual addressing mechanisms included in the operating system. Several research prototypes of SVM, such as Koan [6] and Ivy [8], have demonstrated the feasibility of that

concept.

The SVM programming environment developed at KFA[1] is intended to support scientific programs utilizing SVM on the Intel Paragon. It is based on the Advanced Shared Virtual Memory (ASVM) system, a prototype implementation developed by the Intel European Supercomputer Development Center (ESDC) [9]. ASVM substitutes the XMM layer of the MACH 3.0 micro kernel. The shared address space is implemented through a strong coherency protocol. Data locality is actively supported by page migration. In the case of insufficient memory space on a single node, pages can be paged out to other nodes or to the backing store. On top of ASVM, we are providing SVM-Fortran with a source-to-source compiler.

The use of the SVM programming model leads to a significant shorter and simplier program development cycle. Similar to other high level parallel programming languages, the distance to low-level performance issues is greater than in the message passing program model. Therefore, the user support in program tuning is a critical issue for the future success of SVM.

The performance analysis environment consists of monitoring support integrated into the ASVM, the SVM-Fortran Application Monitor SAM, the source-code-based OPtimizer and locality Analyzer tool OPAL, and the trace visualization tool PARvis. The environment tackles the problem of large trace files by supporting incremental performance analysis. The user can efficiently start from a coarse overview of the performance of his application and can request more detailed information in important code regions within subsequent program runs. Due to the design of the monitoring system, a very specific instrumentation is possible without a recompilation of the program.

In Section 2, we introduce SVM-Fortran, Section 3 gives an overview of the key performance issues of SVM-Fortran, Section 4 outlines the overall performance analysis approach with OPAL, and Section 4.1 describes the concepts of the SVM-Fortran trace format. The performance monitor SAM is presented in Section 4.2, and in Section 4.3 we give an overview of the current implementation status of OPAL and its relations to PARvis. The last section describes our experiences with the tool and gives directions for future work.

2 SVM-Fortran

SVM-Fortran [5] is a shared memory parallel extension of Fortran77 targeted mainly towards data parallel applications on shared virtual memory systems and distributed shared memory systems which provide hardware support for a global address space on top of physically distributed memory. Concepts of HPF, Fortran-S, and KSR Fortran are integrated into a very flexible language, and are adapted to the requirements of programming SVM. The execution model of SVM-Fortran is an extension of the Single-Program-Multiple-Data (SPMD) model which is well-known for its low-overhead parallel execution and is best-suited for hierarchical memory machines.

In addition, SVM-Fortran provides the standard features of shared memory parallel Fortran languages, i.e. shared and private data, multi-dimensional parallel loops and parallel sections, classical synchronization operations as well as SVM-specific synchronization such as variable locking and atomic update. It provides specific features to determine the distribution of parallel work onto processors. Similar to Fortran-S and

[1]More information about the SVM-Fortran project can be accessed via the world wide web: http://www.kfa-juelich.de/zam/PT/ReDec/ProgLangc/SVM.html

```
        SUBROUTINE G(....,T,N,...)
CSVM$   TEMPLATE::  T1(N,N)
CSVM$   PROCESSORS::  P(2,NUMPROC()/2)
CSVM$   DISTRIBUTE (BLOCK,BLOCK) ONTO P::  T1

CSVM$   PDO(LOOPS(I,J),STRATEGY(ON_HOME(T1(I,J))))
        DO I=1,M
              DO J=1,M
                    ...
              ENDDO
        ENDDO
```

Figure 1: SVM-Fortran example program

KSR Fortran, loop annotations can be used to determine a static or dynamic work distribution scheme. Examples are *direct scheduling*, such as BLOCK and CYCLIC, as well as *dynamic scheduling*, e.g. self-scheduling.

Data locality is not a problem which can be solved on the level of individual DO-loops; instead, it is a global problem. Therefore, SVM-Fortran borrowed the concepts of processor arrangements and templates from HPF as tools to specify scheduling decisions globally through template distributions. Template distributions determine the work distribution for parallel loops in *predefined scheduling* which assigns loop iterations to processors according to the distribution of the template.

Templates can be handled in a very flexible way. They can be created dynamically, distributed and redistributed at any point in the program, and passed via the subroutine interface. SVM-Fortran supports standard distributions like BLOCK, CYCLIC, and GENERAL_BLOCK, *indirect distributions* and *linked distributions*.

The example in Figure 1 illustrates the work distribution features of SVM-Fortran. The parallel loop is scheduled via the distribution of template T1.

3 Performance Issues in SVM-Fortran

The key issues in optimizing parallel programs are the reduction of communication and synchronization as well as the improvement of the work distribution. In SVM systems communication occurs in form of page faults. If a shared variable or array is accessed by a processor and the corresponding page is not available in the processor's memory, a pagefault occurs. This pagefault is serviced by another processor which has a valid copy of that page.

In some situations, the value of the variable was written by one processor and the faulting processor uses it. This situation is called true-sharing in contrast to false-sharing. False-sharing occurs due to the large page size of 8 Kbytes in the ASVM. Several array elements or scalar variables are located on the same page. If a processor writes a variable on the page, the processor invalidates all other copies of the page due to the strong coherency protocol. If another processor reads a value of another variable on the same page, the page has to be transferred. This communication is unnecessary and therefore it is called false-sharing.

If a page is frequently moved back and forth between two processors in a short time frame due to true- or false-sharing this is called page thrashing. Page thrashing is the worst situation with respect to communication.

In addition to page faults, the synchronization overhead is a main reason for performance degradation. In SVM-Fortran, each parallel region has two barriers to implement the standard semantics. The user can eliminate the synchronization by using the NOBARRIER option of parallel loops and parallel sections. Since eliminating barriers is a busy transformation, it is very useful for the user to understand the overhead resulting from synchronization. He will only start optimizing synchronization if the potential gain is worthwhile.

In contrast to other parallel languages, the user can specify the scheduling strategy explicitly, for example iterations of parallel loops are scheduled block-wise onto the processors. The user's goal is to find an ideal compromise between optimal load balance and absolutely no page faults. Ideal load balance in general requires an equal number of iterations for each processor when distributing a parallel loop. On the other hand, the iterations should be distributed among the processors in the way that a processor executes all iterations accessing the same page. Therefore, a performance analysis tool has to give information to the user such that he is able to choose a good compromise.

In addition to the requirements imposed by the SVM programming model some general demands have to be fulfilled by a performance analysis tool. The intrusion due to the monitoring should be in a range, that performance values are still reliable. If this cannot be reached, information about the monitoring overhead should be available to the user.

As already stated in the introduction, the presentation of performance values should include information on user level, e.g. source code related information. Another desirable feature is some form of user guidance in order to support the user in finding the most important bottlenecks. Last but not least, the tool should provide a framework that controls the optimization process and administrates the different program versions with their performance information files.

4 OPAL

OPAL is a performance analysis and optimization tool that integrates program text and runtime performance information to support the user in optimizing data locality. Figure 2 illustrates the performance analysis features of OPAL and its interfaces to the other components of the programming environment.

OPAL interacts with the monitoring subsystem via a performance analysis database. The overall performance analysis process is incremental, i.e. starting from a coarse overview of the runtime behavior, more detailed information for specific code regions is gathered in subsequent program runs.

The concept of incremental performance analysis was chosen to reduce the amount of monitoring data and thus to get around the monitoring impact on the application. Central to this concept are a hierarchically structured trace format and an efficient implementation of selective tracing.

The trace records form a hierarchy which is described here for the page travel information. Page travel information is the main source for analyzing data locality in SVM systems. Page faults occur when data is accessed that is placed on a page currently

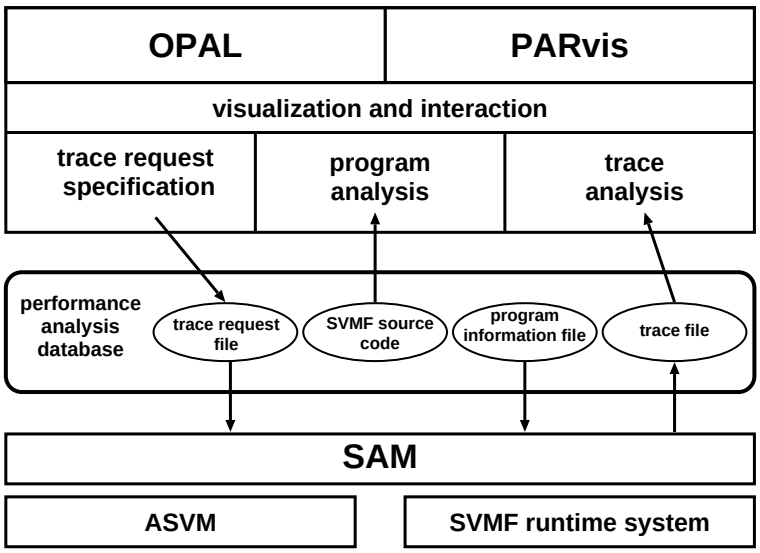

Figure 2: Performance analysis environment for SVM-Fortran

mapped in the memory of another processor. Tracing all page faults would generate an enormous amount of data. Therefore, a hierarchy of trace records is available. The low level records trace individual page faults and other page movements, such as page out operations if the memory of a processor is filled up. Above that level, a page travel record is available that counts for each processor the number of pages received from that processor. The coarsest information are pagefault sums. These sums describe how many page faults have occured. All the information can be traced for arbitrary address regions like shared arrays etc.

As an example for the usage of this information, we refer to page thrashing resulting from false-sharing. This situation can be identified if all page faults are traced. A good hint to identify page thrashing is a high number of page faults already traced by page-fault sums. To detect that page thrashing occurs, the page travel event can be used, which gives more information. For example, when analyzing the page travel events of two processors, page thrashing is very likely, when the sum of pages, received from the respective processor is in the same range.

Selective tracing is implemented through a *trace request file* that contains the information which events have to be monitored. When the program is compiled for performance analysis, hooks to the monitor SAM are inserted in the source code. In the consequence, the source has to be compiled only once, and the user can control the monitoring through changes in the trace request file.

4.1 SVM-Fortran Trace Format

The SVM-Fortran trace format is based on the Self Defining Data Format (SDDF) [2]. It includes records for tracing page travel, profiling, synchronization, overhead, and language information.

All trace records include source code information. We trace the location in the source via a numbering scheme for regions, directives and variables. The source code information is created by the SVM-Fortran compiler.

In addition to other basic information such as synchronization events, we also pro-

vide records for tracing language information. Due to the intrusion of dynamic tracing on the program, as much analysis as possible should be done based on static program information. This approach, usually called performance prediction, suffers from any runtime dependent information, such as variables in loop boundaries etc. Therefore, we can trace information needed to make performance prediction more accurate. For example, the distribution of dummy templates may not be known from the source code but is needed to predict the behavior of loops with predefined scheduling. Therefore, the mapping of these templates onto templates for which the distribution is known has to be traced.

4.2 Monitoring support

The performance monitor SAM [10] collects data that is needed for the performance analysis in OPAL. When the program is started, the monitor reads the trace request file which includes trace requests such as

REQUEST (∗) LOCAL foo.PDO_LABEL(150): RPF (A), WPF (A)

specifying that for all processors (∗) read and write pagefault events should be generated for array `A` during execution of the parallel DO-loop with label 150 in subroutine `FOO`. The keyword LOCAL starts a trace request for a parallel region, e.g. subroutine, or parallel loop. GLOBAL introduces requests for tracing events in the whole program, like monitoring or SVM overhead. These request are generated automatically by OPAL on user demand.

SAM translates program symbols like labels and variables with the help of the program information file generated by the SVM-Fortran compiler into runtime values. According to the trace requests, SAM collects the information from the appropriate runtime data structures and requests SVM-specific information like pagefault information from the ASVM. The ASVM provides a very flexible interface which supports the collection of data for arbitrary address regions. The tracing of language events will be implemented in the next release of OPAL and SAM.

4.3 Performance Analysis

In the following we describe the general performance analysis concept for SVM-Fortran programs and the practical use of OPAL when optimizing a SVM-Fortran program.

Figure 3 gives a detailed overview of the performance analysis cycle. The user can start by specifying trace regions via directives (A). Trace regions are code sections for which specific runtime information can be requested. Most code sections, the user might be interested in, are trace regions by default, such as subroutines and parallel loops. If the user does not want to specify trace regions he can directly start by compiling the code for performance analysis with the SVM-Fortran compiler and the native compiler of the target system (B).

In the second phase, trace requests are specified for individual trace regions in a trace request file which serves as an input to the execution and guides runtime tracing. During execution, events are written to a trace file according to the trace requests.

Trace events are used in the analysis step in combination with information gathered from the source code to identify performance bottlenecks. The compiler's mapping information is needed in this step to correctly relate trace information to the source code.

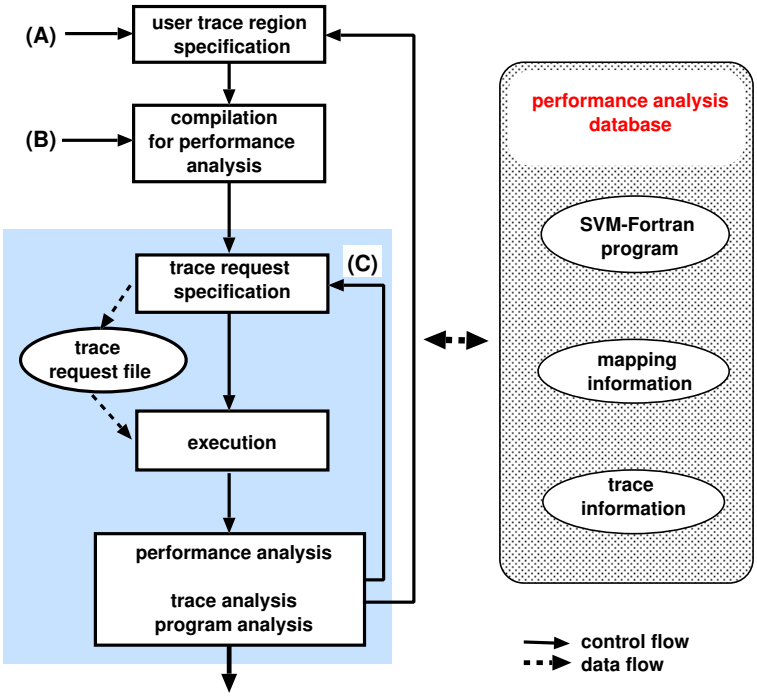

Figure 3: Performance analysis with OPAL

If performance bottlenecks cannot be identified precisely with the available information the trace request file can be adapted (C) and more information be gathered without recompilation. All the information used in the performance analysis is combined in a database.

OPAL assists the user in all performance analysis phases on the basis of the source code related view. Units are accessible through a name list in a separated window, parallel regions can be truncated so that only the CSVM$-directives are visible, and comments can be hidden. These techniques allow the user to achieve a good overview of his program code.

OPAL supports a menu-driven specification of trace requests that frees the user from the subtle specification syntax. In correspondence with the requirements pointed out in Section 3 the user can formulate requests for

- read/write pagefault information: Counter, traces for regions and for specific arrays are selectable
- synchronization information
- overhead information of the performance monitoring and the SVM-Fortran run-time system
- information on flushing the trace buffer

This can be done for interactively marked regions, or for the whole program (in this case only counters will be requested, since the trace file would exceed a handleable size).

In the third step, the individual trace files of the processors are merged and read by OPAL. OPAL extracts the most useful statistical data, but does not store all the trace information in memory. These statistical data can be visualized as annotations to the source code in a separate performance column in the main window.

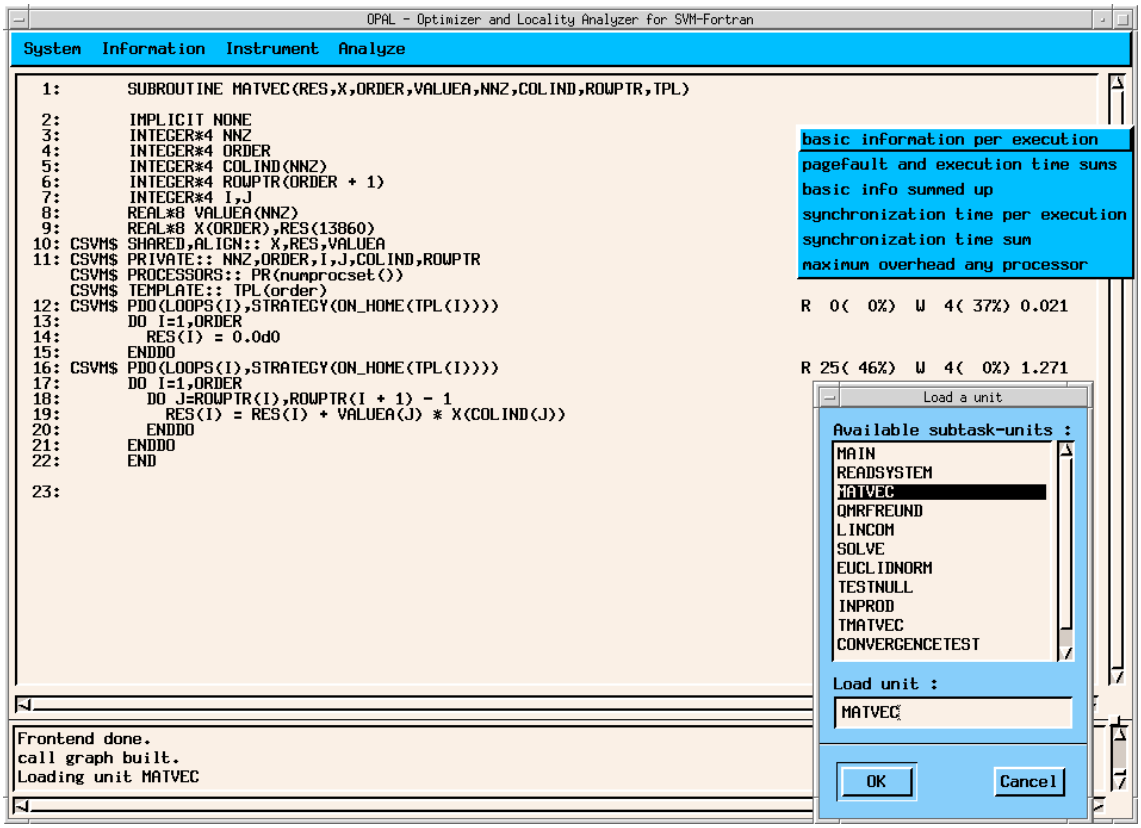

Figure 4: Source code related information in OPAL

The user can request the following types of information in the performance column via a popup menu:

- basic information
  The most basic information are the number of read and write page faults in a trace region and the execution time. This information can be presented as mean values for an individual execution of the trace region or summed up over all executions. The percentages determine the portion of time spent in waiting for pages relative to the execution time.

- pagefault time
  The pagefault time summed up for all executions of the trace region.

- synchronization time
  The synchronization time can be presented for individual executions of the trace region or summed up over all executions.

- maximum overhead
  The individual overheads, i.e. pagefault time and synchronization time, are presented in form of mean values for all processors. But, if one processor has a high pagefault time, the other processors will have high synchronization times because of the barrier at the end of the region. Both overheads are summed up and the maximum value is displayed by OPAL. The maximum overhead is determined in OPAL as the maximum of the processor specific sum of both overheads.

If the user needs more information on individual trace regions, he can mark a trace region and request a list of all trace events of a specific type. For example, OPAL presents all shared variables accessed in the selected trace region and the user can select specific variables for which he would like to see individual page faults. The tool then extracts from the trace file of an individual processor only the pagefault information for these variables and the selected program region. Due to the source code information in the trace files, all the information can be requested and presented in form of trace regions and program variables, e.g. the faulting address is translated into the array name and the indices of the array element.

We also implemented a limited from of user guidance. Similar to the well-known tool gprof the user has the possibility to request a separate window with a sorted time summary for all units and a sorted list of the maximum number of read and write page faults in the program. Furthermore, the most interesting code sections can be accessed in the main window by simply clicking on the displayed values in the overview windows. In this way the user gets a hint to decide where performance debugging is most advisable.

The experience made with OPAL has shown that page faults due to nonoptimal work distribution, alignment, and false-sharing in single procedures are handled well by the tool. Remaining page faults due to accesses in different subroutines with changing names and different control flow need to be analyzed by looking at the dynamic behavior of the code.

On the one hand, this is a field of research that includes new analysis techniques for tracking page faults over subroutine boundaries and rule-based bottleneck detection. Moreover, predictions about the effectiveness of the user's efforts in analyzing specific code regions would lead to an efficient optimization procedure.

On the other hand, the use of the trace visualization tool PARvis, also developed at our institute [1], could give more insight into irregular performance behavior. PARvis is part of an integrated tool environment to visualize system activities by processing trace event files. It includes state-of-the-art visualization techniques and allows trace analysis on different granularity via sophisticated zooming and flexible scrolling techniques. Currently, the interface between OPAL and PARvis is being developed to connect the source code related view with the trace visualization techniques [3].

5 Conclusions and Future Work

In this article we described the parallel programming environment of SVM-Fortran. The whole environment is used for half a year for parallelizing applications in SVM-Fortran. It turned out that the use of the performance analysis tool OPAL was absolutely necessary in the parallelization process [4]. It was only possible to understand the performance bottlenecks of the codes by using this tool showing the page faults and the synchronization overhead. After identifying page faults for an array, it is important to understand where the page faults occur and where the pages or the permissions are lost. This is supported quite well in OPAL since all paging information for an individual array can be requested.

Naturally not all requirements have been met by OPAL yet. Further work has to be done. Special emphasis will be set on the realization of performance debugging demands that are rarely implemented in existing performance tools [7]. This includes:

- More user guidance for the detection of performance bottlenecks
- Automatic fixing of performance bottlenecks
- Providing different program views for instrumenting the program
- Integration of the iterative optimization process in the tool
- Support for programming irregular problems
- Emphasizing the cache performance of the parallelized code

These issues together with a global concept for optimizing locality in SVM-Fortran programs will be an important step towards reducing the performance gap between the time consuming message passing programming style and the easy-to-use shared virtual memory approach.

References

[1] A. Arnold, U. Detert, W.E. Nagel, *Performance Optimization of Parallel Programs – Tracing, Zooming, Understanding,* 35. Semi-Annual Cray User Group Meeting, Denver, pp. 252-258, 1995

[2] R.A. Aydt, *The Pablo Self-Defining Data Format,* Department of Computer Science, University of Illinois, Urbana, Illinois 61801, 1994

[3] J. Bernert, *Neue PARvis Komponenten zur effizienten Performance-Analyse von SVM-Fortran-Programmen,* Diploma thesis, RWTH Aachen – Research Centre Jülich (KFA), 1995

[4] R. Berrendorf, M. Gerndt, *Parallelizing Applications with SVM-Fortran,* Proc. of the HPCN'95, Milano, LNCS 919, pp. 793-798, 1995

[5] R. Berrendorf, M. Gerndt, *SVM-Fortran Reference Manual Version 1.4,* Internal Report KFA-ZAM-IB-9510, Central Institute for Applied Mathematics, Research Centre Jülich (KFA), 1995

[6] F. Bodin, T. Priol, *Overview of the Koan Programming Environment for the iPSC/2 and Performance Evaluation of the BECAUSE Test Program 2.5.1.,* Proc. of the BECAUSE European Workshop, Sophia-Antipolis, 1992

[7] A. Hondroudakis, *Performance Analysis Tools for Parallel Programs,* Edinburgh Parallel Computing Centre, The University of Edingburgh, 1995

[8] K. Li, *IVY: A Shared Virtual Memory System for Parallel Computing,* Proc. 1988 Int. Conf. on Parallel Processing, Vol II, pp. 94-101, 1988

[9] M. Mairandres, S. Zeisset, *Scalable and Efficient Management of Pages in Shared Virtual Memory Systems,* Intel Supercomputer Development Center (ESDC), Feldkrichen/Munich, 1994

[10] S. Özmen, *SAM: Performance-Analyse-Monitor für SVM-Fortran,* Diploma thesis, RWTH Aachen – Research Centre Jülich (KFA), 1995